

CSE687 Midterm #3

Name: _____ Instructor's Solution_____

This is a closed book examination. Please place all your books on the floor beside you. You may keep one page of notes on your desktop in addition to this exam package. All examinations will be collected promptly at the end of the class period. Please be prepared to quickly hand in your examination at that time.

If you have any questions, please do not leave your seat. Raise your hand and I will come to your desk to discuss your question. I will answer all questions about the meaning of the wording of any question. I may choose not to answer other questions.

You will find it helpful to review all questions before beginning. All questions are given equal weight for grading, but not all questions have the same difficulty. Therefore, it is very much to your advantage to answer first those questions you believe to be easiest.

1. Give definitions for the terms "overload" and "override" as used by the C++ language. When would you use overloading, when overriding, and why?

Answer:

Overload refers to two or more functions with the same name in the same scope but with different calling arguments. C++ distinguishes them by creating at compile time "mangled names" which are a tokenized version of the concatenation of function name and each of the types of its arguments in the order declared for the function. Thus, overloaded functions are simply different functions to the compiler.

We use overloads when two or more functions have the same type of operation, but using different parameters, e.g., constructors, display functions, etc.

Override refers to Derived class functions which redefine virtual base class functions. Overrides must use exactly the same function signature as the overridden base function, with the exceptions that the return type may be covariant¹ and the use of the virtual qualifier in the derived class is optional, but desirable. Overrides are resolved² at run-time using each class's virtual function pointer table.

Overrides allow a derived class to respond to function calls declared in the base protocol with derived class specific behavior. If we think of a base class as providing a generic model for the behavior of a type then derived class overrides allow us to specialize that model to a set of behaviors needed for a specialized type.

¹ Covariant means that if the base function returns a pointer to the base type the derived function may return a pointer to the derived type.

² Here, resolved means deciding which function to call.

2. For the XmlDocument class, in the Project #2 helper code, write all the code for a function size() that returns the number of elements in the parse tree excluding text nodes by visiting each node. Describe how you would change the function to cache the count so the node visitation is not required for every invocation of size().

Answer:

```
//-----< find element(s) with this tag >-----
/*
    if tag == "" returns pElem and all decendents
*/
bool XmlDocument::find(const std::string& tag, sPtr pElem, bool findall)
{
    if (pElem->tag() == tag || tag == "")
    {
        found_.push_back(pElem);
        if (!findall)
            return true;
    }
    for (auto pChild : pElem->children())
        find(tag, pChild);
    return (found_.size() > 0);
}
//-----< return the number of elements in the document >-----

size_t XmlDocument::size()
{
    find("", pDocElement_, true);
    size_t size_ = found_.size() - 1; // don't count docElement
    found_.clear();
    return size_;
}
```

If the class provides a cache variable, say size_, perhaps initialized to -1, then after the first call we increment or decrement the cache value with every addChild and removeChild. Note that we still have to compute the size of the added or removed element, so caching may not save a lot of computation.

So, when size() is called we check size_ and if not -1 we return size_, otherwise we compute as above.

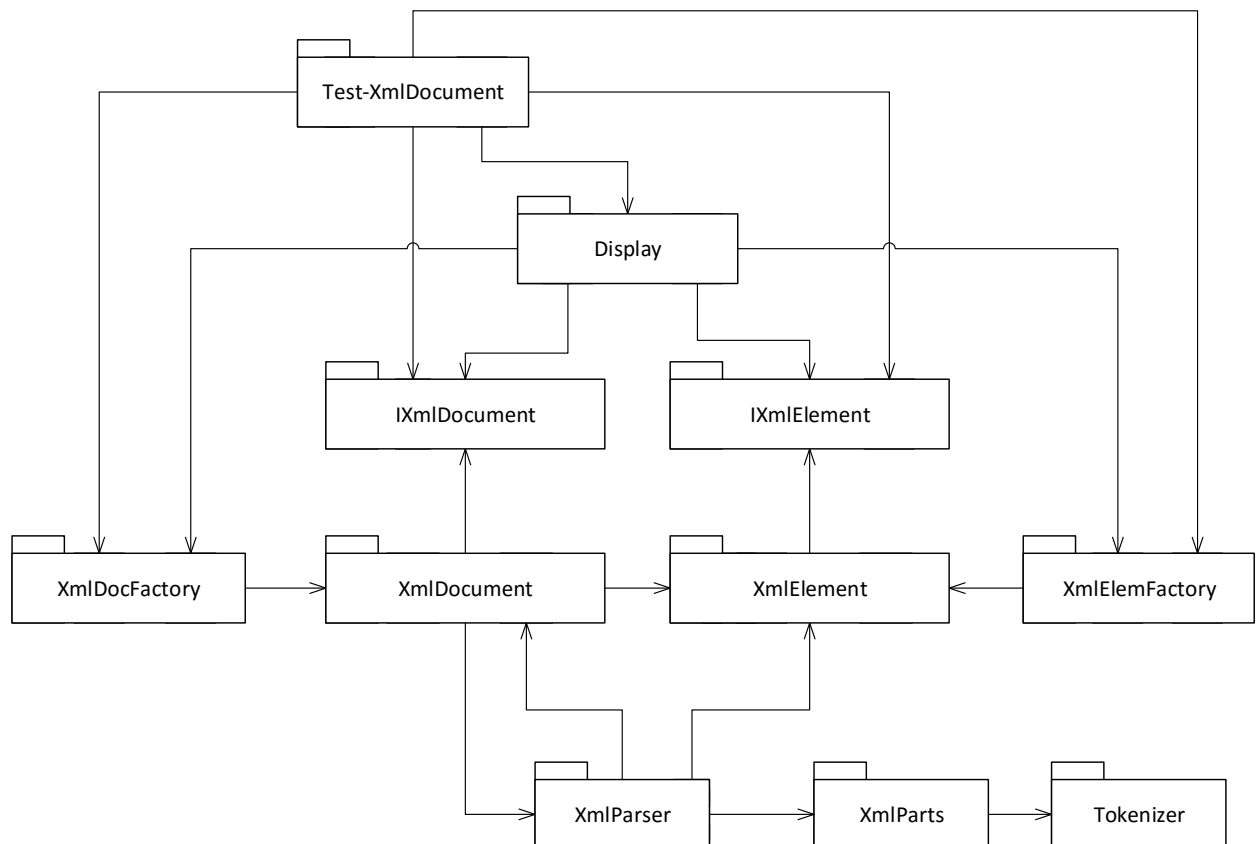
3. State the Dependency Inversion Principle and discuss how it applies to your design of Project #2.

Answer:

"High level components should not depend upon low level components. Instead, both should depend on abstractions."

"Abstractions should not depend upon details. Details should depend upon the abstractions."

The AbstractXmlElement class provides an abstraction for clients to use and partially buffers them from interacting with concrete types, as do the makeElement factories. However, for complete dependency inversion we need to provide interfaces and object factories like this:



4. Provide all the code for a global Depth First Search function that operates on a tree of instances derived from the AbstractXmlElement class and accepts a callable object to operate on each element found during the search. Write all the code for a callable object that displays on the console the tag of each tagged element encountered. How would you modify the AbstractXmlElement class to allow the callable object to perform that function without knowing the derived type of the elements encountered during search?

Answer:

In order to handle tags polymorphically we need to add a member function:

```
virtual std::string tag() { return ""; }
```

to the AbstractXmlElement class and override that in the TaggedElement class to return its tag.

```
//-----< search subtree of XmlDocument >-----

template<typename CallObj>
void DFS(XmlDocument::sPtr pElem, CallObj& co)
{
    using sPtr = XmlDocument::sPtr;
    co(*pElem);
    for (auto pChild : pElem->children())
        DFS(pChild, co);
}

//-----< test Depth First Search on XmlDocument >-----

void testDFS(XmlDocument& doc)
{
    title("testing DFS on XmlDocument - printing tags");

    std::function<void(AbstractXmlElement&)> f; // callable object
    f = [](AbstractXmlElement& Elem) {
        if (Elem.tag().size() > 0)
            std::cout << "\n " << Elem.tag();
    };

    DFS(doc.pDocElement_, f);
    std::cout << "\n";
}
```

The DFS function used here is very similar to the TaggedElement::toString() method discussed in class. Both use recursive calls to walk the AST.

5. What is move assignment? Write a code fragment that will cause a move assignment for an instance of the `std::string` class. Show enough of the code context to see why the move operation will be called.

Answer:

Note:

The discussion of Move in the design note C++11 incorrectly used `const` in the argument³. If you've done that I marked it wrong but did not deduct any points for that.

Move assignment is an overload of the assignment operator with the signature:

```
T& operator=(T&& t); // no const here
```

where `T` is some specified type. Its function is to efficiently transfer ownership of the source's resources to the target, resulting in the source holding an empty or invalid resource. In the declaration above, the source is `t` and the target is the object invoking the assignment operator, e.g., the object referenced by the "this" pointer.

Often a move is implemented by copying a pointer to the source's resources to the target's resource pointer, not by copying the resource. Then the source's resources are deallocated. Note that the contents don't move. Only the resource reference is copied.

```
std::string doMove()
{
    std::string temp = "this is a moved test string";
    return temp;
}

int main()
{
    title("MT3.Q5 - Demonstrating string move");
    std::string movedStr;
    movedStr = doMove();
    std::string anotherMove;
    anotherMove = movedStr + " extended by a second move";

    std::cout << "\n " << anotherMove << "\n\n";
}
```

³ Now fixed.

6. What is a template trait? Give an example for one of the STL containers. Why do the STL containers provide traits?

Answer:

A template trait is a typedef or alias (using statement) that provides an invariant name for a template type that will change when an application instantiates the template with a specific type. For `std::list<T>` the trait is:

```
typedef T value_type; or using value_type = T;
```

```
std::list<std::string> => value_type is std::string
```

Traits are necessary to write template functions that operate on template classes and need to use types exposed by the template class before an application instantiates the class.

For example, to sum the values of a container whose type is undefined at the time we write the function we will have to declare the sum variable which depends on a type we do not know. But if the container provides a `value_type` trait we can declare the sum like this:

```
C::value_type sum = value_type();
```

If `C` is `std::vector<int>` then `C::value_type` is `int`. If `C` is `std::list<double>` then `C::value_type` is `double`.

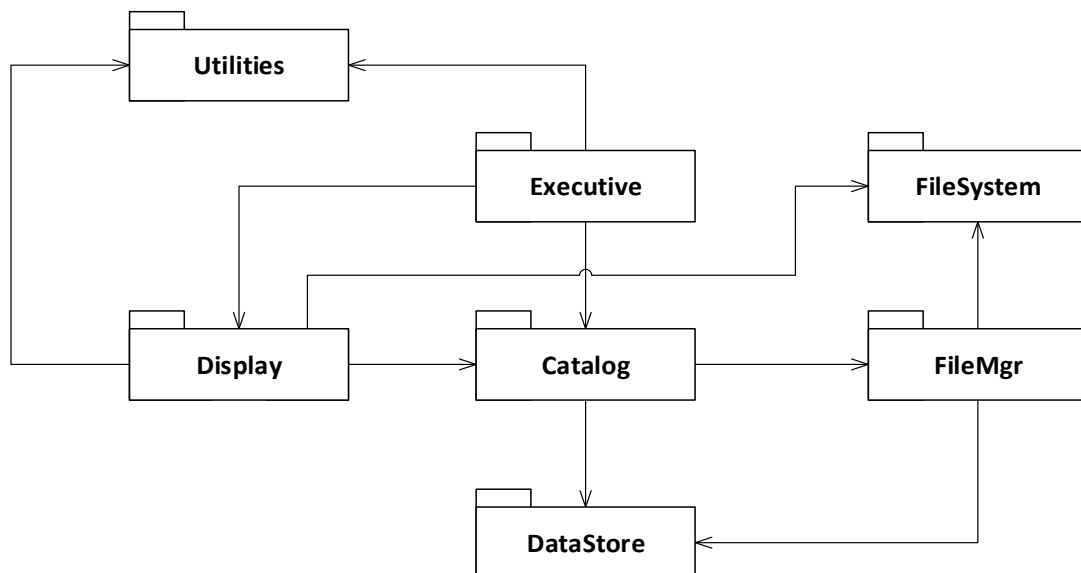
7. Discuss how you designed Project #1 to support communication between all packages that need to communicate. Support your answer with a package diagram, discussing when communication occurs and what is communicated.

Answer:

The Executive creates an instance of Catalog, passing the path, and creates an instance of Display, passing the commandline and a pointer to Catalog to its Display instance, as constructor arguments. Display checks for the existence of the path passed from Executive using the services of FileSystem

The Catalog class composes instances of DataStore and FileMgr. Its constructor creates an instance of DataStore and passes that, along with the analysis path, to FileMgr as constructor arguments in Catalog's constructor initialization sequence. That enables all the communication needed between these three classes.

This completes setting up all communication necessary for operation of Project #1.



// End of answer

////////////////////////////////////

// More detail not expected in your answer but useful for those trying to
// understand how the project works:

In my design the Executive and Display drive the entire processing flow. Executive passes its commandline to display for parsing. Display is given the responsibility of parsing the commandline because that's where all the information about what to display resides.

The Display instance passes file patterns to Catalog through `Catalog::addPattern(...)` which in-turn passes the same patterns to `FileMgr::addPattern(...)`. Executive then instructs Catalog to build its internal data.

`Catalog::buildCatalog(...)` simply requests `FileMgr` to search, which results in the `DataStore` being filled with directory information. As `FileMgr` finds each file it writes the file information into the `DataStore` instance, avoiding holding duplicate filename and path information.

When that process is finished the Executive instructs Display to show all the results that match its parsed options. Catalog makes that data available to Display through member functions `showFiles()`, `showDuplicates()`, `showSearchResults()`, and `showSummary()`.

Note that Executive doesn't need to know anything about the options and patterns and is not a conduit for all the program's data. That could cause a lot of unnecessary copying.

